

CS1010S Programming Methodology

Lecture 9

Generic Operations

25 March 2015

Agenda

- Recap: Complex-Arithmetic Package
- Tagging Data
- Implementing Generic Operators
 - Dispatch on Type
 - Data-Directed Programming
 - Message Passing

Complex-arithmetic package

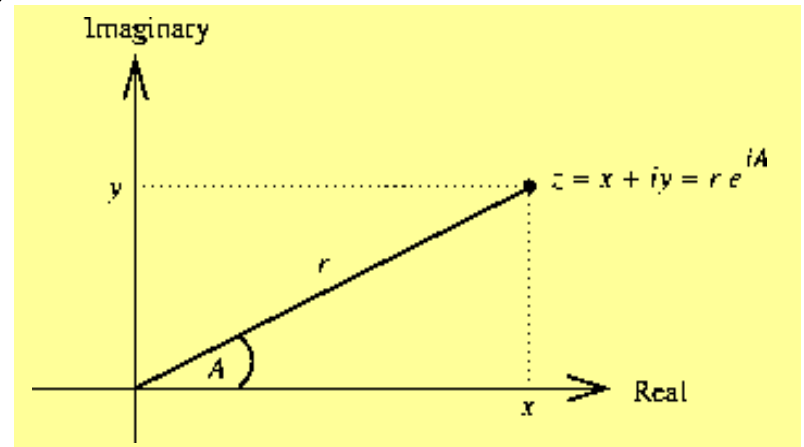
- Recall: complex numbers

$$3 + 4i, i = \sqrt{-1}$$

$x + iy$, x : real part, y : imaginary part

- The above is **rectangular** form.
- There is also **polar** form:

- $z = x + iy = r e^{-iA}$
- r : magnitude
- A : angle



Abstraction barrier

Programs that use complex numbers

<code>add_complex, sub_complex, mul_complex, div_complex</code>

Complex Numbers Package

Rectangular representation	Polar representation
-------------------------------	-------------------------

Arithmetic package

Similar to rational number package (Lecture 7)

Addition: $z = z1 + z2$

$$\text{real_part}(z) = \text{real_part}(z1) + \text{real_part}(z2)$$

$$\text{imag_part}(z) = \text{imag_part}(z1) + \text{imag_part}(z2)$$

Arithmetic package

Multiplication: $z = z1 * z2$

$\text{magnitude}(z) = \text{magnitude}(z1) * \text{magnitude}(z2)$

$\text{angle}(z) = \text{angle}(z1) + \text{angle}(z2)$

etc.

These are easily implemented assuming the existence of

selectors: `real_part`, `imag_part`,
`magnitude`, `angle`

constructors: `make_from_real_imag`, `make-from-mag-ang`

Wai Kay's (rectangular) code

```
def make_from_real_imag(x, y):  
    return (x, y)  
def real_part(z):  
    return z[0]  
def imag_part(z):  
    return z[1]  
  
def magnitude(z):  
    return math.hypot(imag_part(z),  
                      real_part(z))  
def angle(z):  
    return math.atan(imag_part(z) / real_part(z))  
  
def make_from_mag_ang(r, a):  
    return (r * math.cos(a), r * math.sin(a))
```

Code in Action

```
def print_complex(z):  
    print(str(real_part(z)) + " + "  
          str(imag_part(z)) + "i")
```

```
a = make_from_real_imag(1, 1)  
b = make_from_real_imag(2, 2)  
print_complex(a) → 1 + 1i  
print_complex(b) → 2 + 2i
```

```
def add_complex(z1, z2):  
    return make_from_real_imag(  
        real_part(z1) + real_part(z2),  
        imag_part(z1) + imag_part(z2))
```

```
print_complex(add_complex(a, b)) → 3 + 3i
```


Karen's (polar) code

```
def make_from_mag_ang(r, a) :  
    return (r, a)  
  
def real_part(z):  
    return magnitude(z) * math.cos(angle (z))  
  
def imag_part(z):  
    return magnitude(z) * math.sin(angle(z))  
  
  
def magnitude(z):  
    return z[0]  
  
def angle(z):  
    return z[1]  
  
def make_from_real_imag(x, y):  
    return (math.hypot(x, y), math.atan(y / x))
```

Same Code!!

```
a = make_from_real_imag(1, 1)
b = make_from_real_imag(2, 2)
print_complex(a) → 1.000000000000000002 + 1.0i
print_complex(b) → 2.000000000000000004 + 2.0i
```

```
def add_complex(z1, z2):
    return make_from_real_imag(
        real_part(z1) + real_part(z2),
        imag_part(z1) + imag_part(z2))
print_complex(add_complex(a, b))
→ 3.000000000000000001 + 3.0i
```

Two Lessons

- Different representations might have slightly different accuracy because of internal representation
- Power of data abstraction
 - Can reuse same code even though the representation is completely different

Co-existing Multiple Representations:

What are the issues?

Rect version:

```
make_from_real_imag(x, y)
real_part(z)
imag_part(z)
magnitude(z)
angle(z)
make_from_mag_ang(r, a)
```

Polar version:

```
make_from_real_imag(x, y)
real_part(z)
imag_part(z)
magnitude(z)
angle(z)
make_from_mag_ang(r, a)
```

(2, 2)
Rect

(4, 9)
Rect

(2, 3)
Polar

But they all
look the same!

(4, 1)
Polar

(1, 3)
Rect

(7, 3)
Rect

All tuples!!

Co-existing Multiple Representations

1. Matching representations to operations.

- e.g. **rectangular rep.** must be given to **rectangular functions**.
- Polar rep. cannot be given to rectangular functions, or vice versa.

Co-existing Multiple Representations

2. Name conflicts

- Both reps have functions with identical names!
- e.g. `real_part`, `angle`

Matching rep. to operations

- Solution: tagged data
 - Each representation is given a tag to explicitly indicate the representation (type).
 - e.g. rectangular (<rectangular data>)
 - e.g. polar (<polar data>)

Tagging

```
def attach_tag(type_tag, contents):  
    return (type_tag, contents)
```

```
def type_tag(datum):  
    if type(datum) == tuple and  
        len(datum) == 2:  
        return datum[0]  
    else:  
        raise Exception('Bad tagged datum --  
type_tag' + str(datum))
```



Exception: Signals something
bad has happened

Tagging

```
def contents(datum):  
    if type(datum) == tuple  
        and len(datum) == 2:  
        return datum[1]  
    else:  
        raise Exception('Bad tag datum  
-- contents' + str(datum))
```

Matching rep. to operations

Wai Kay's revised constructor:

```
def make_from_real_imag(x, y):  
    return attach_tag('rectangular',  
                      (x, y))
```

Tagged data

Karen's revised constructor:

```
def make_from_real_imag(x, y):  
    return attach_tag('polar',  
                      (math.hypot(x, y),  
                       math.atan(y/x)))
```

Tagged data

To check for tag:

```
def is_rectangular(z):  
    return type_tag(z) == 'rectangular'
```

```
def is_polar(z):  
    return type_tag(z) == 'polar'
```

Name conflicts

But Wai Kay's and Karen's
functions still have the
same name!

Whoops!

Name conflicts

Solution: impose naming convention function name ends with tag

```
def make_from_real_imag_rectangular(x, y):  
    return attach_tag('rectangular', (x, y))
```

```
def make_from_real_imag_polar(x, y):  
    return attach_tag('polar',  
                      (math.hypot(x, y),  
                       math.atan(y/x)))
```

Have we solved
the problem?

Whose responsibility?

- **Sanity check:** whose responsibility is it to match data type with the corresponding operations?
 - User ?
 - Wai Kay?
 - Karen?
 - Other?

Let's Make it the User's Problem?

Programs that use complex numbers

`add_complex, sub_complex, mul_complex, div_complex`

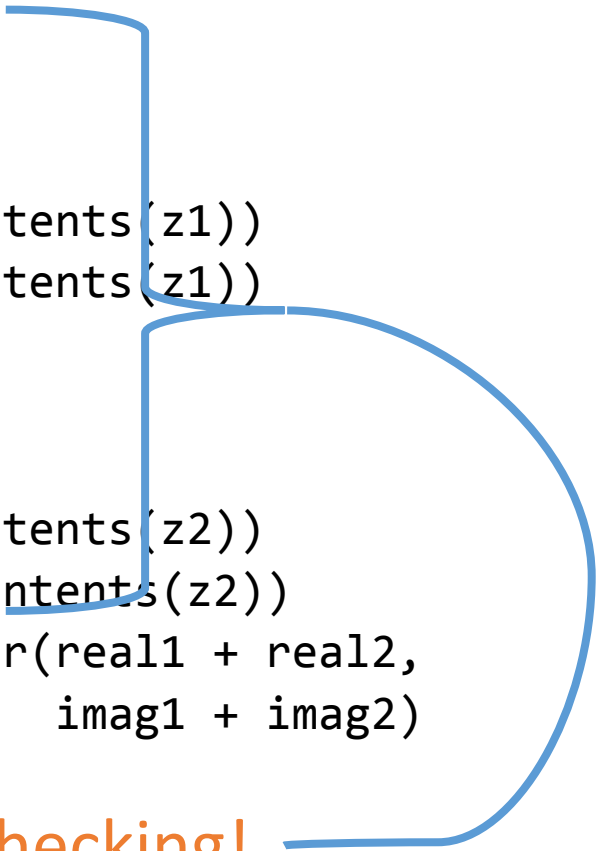
Complex Numbers Package

Rectangular representation	Polar representation
-------------------------------	-------------------------

User's responsibility

User needs to check for rep. and call appropriate functions:

```
def add_complex(z1, z2):  
    real1 = real_part_polar(contents(z1))  
    imag1 = imag_part_polar(contents(z1))  
    if is_rectangular(z1):  
        real1 = real_part_rectangular(contents(z1))  
        imag1 = imag_part_rectangular(contents(z1))  
    real2 = real_part_polar(contents(z2))  
    imag2 = imag_part_polar(contents(z2))  
    if is_rectangular(z2):  
        real2 = real_part_rectangular(contents(z2))  
        image2 = imag_part_rectangular(contents(z2))  
    return make_from_real_imag_rectangular(real1 + real2,  
                                            imag1 + imag2)
```



A lot of code devoted to checking!

Wai Kay's (rectangular) code

```
def make_from_real_imag_rectangular (x, y):  
    return (x, y)  
def real_part_rectangular(z):  
    return z[0]  
def imag_part_rectangular (z):  
    return z[1]  
  
def magnitude_rectangular (z):  
    return math.hypot(imag_part(z),  
                      real_part(z))  
def angle_rectangular (z):  
    return math.atan(imag_part(z) / real_part(z))  
  
def make_from_mag_ang_rectangular (r, a):  
    return (r * math.cos(a), r * math.sin(a))
```

Karen's (polar) code

```
def make_from_mag_ang_polar(r, a) :  
    return (r, a)  
  
def real_part_polar(z):  
    return magnitude(z) * math.cos(angle(z))  
  
def imag_part_polar(z):  
    return magnitude(z) * math.sin(angle(z))  
  
def magnitude_polar(z):  
    return z[0]  
  
def angle_polar(z):  
    return z[1]  
  
def make_from_real_imag_polar(x, y):  
    return (math.hypot(x, y), math.atan(y / x))
```

Whither the future?

- One or more of the following *may* happen in the future:
 - Wai Kay's code is removed. Only Karen's polar representation is left.
 - A new representation for complex numbers is installed, and co-exists with Wai Kay's and Karen's.
 - Karen provides a new function for her rep. that no one else can provide.
- Who should now be responsible, so as to minimize the effects of the above changes?

MANAGING COMPLEXITY

Generic Operators

- Create another layer of abstraction.
 - To provide generic operators.
 - To shield user from the complexity of managing multiple representations.
- Wai Kay or Karen or user can create this layer. Or the project leader.

Abstraction barrier

Programs that use complex numbers

```
add_complex, sub_complex, mul_complex, div_complex
```

Complex Numbers Package

Rectangular representation	Polar representation
-------------------------------	-------------------------

New Abstraction Layer

Programs that use complex numbers

```
add_complex, sub_complex, mul_complex, div_complex
```

Complex Numbers Package

Generic Operators

Rectangular
representation

Polar
representation


Add new layer



The diagram illustrates the addition of a new abstraction layer to the Complex Numbers Package. It features a table with two columns: 'Rectangular representation' and 'Polar representation'. Above the table is the text 'Generic Operators'. An orange arrow points from the text 'Add new layer' to the 'Generic Operators' text, indicating the addition of a new layer of abstraction.

Generic operators

User sees this generic operator.



```
def real_part(z):  
    if is_rectangular(z):  
        return real_part_rectangular(  
                                                    contents(z))  
  
    elif is_polar(z):  
        return real_part_polar(contents(z))  
  
    else:  
        raise Exception('Unknown type --  
                           real_part' + z)
```

Type checking, tag removal, are hidden from user.

Generic operators

- User's code:

```
def add_complex(z1, z2):  
    return make_from_real_imag(  
        real_part(z1) + real_part(z2),  
        imag_part(z1) + imag_part(z2))
```

- Greatly simplified!
- Selectors are generic operators.
- Constructor: we can choose the best one
 - Use rectangular when we have real-imag.
 - Use polar when we have magnitude-ang.
- How do we implement this?

Strategy #1: Dispatching on type

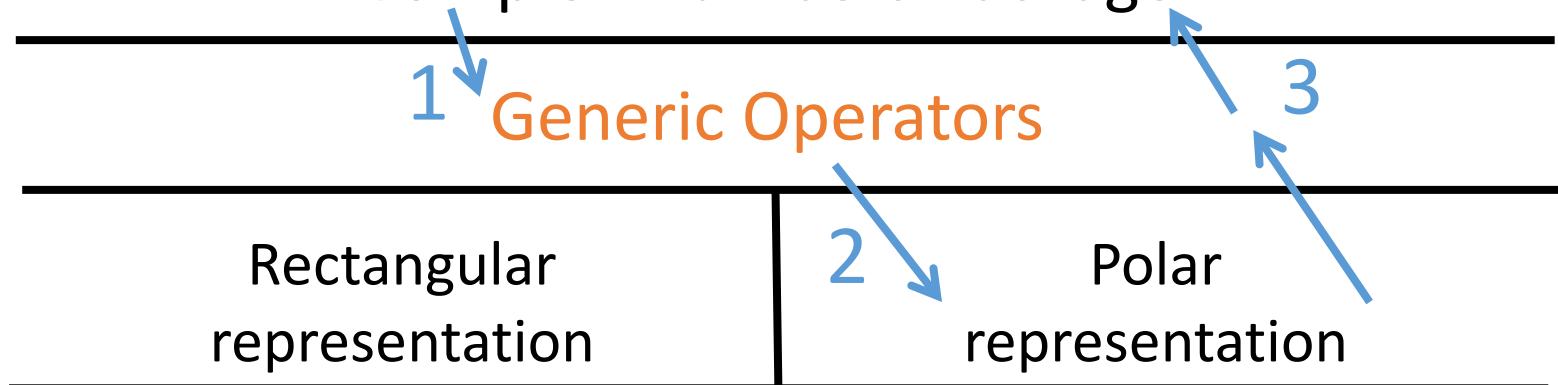
- This strategy of providing generic operators based on checking the data type and calling the appropriate function is called *dispatching on type*.
- Generic operators work on data that could take on *multiple forms* (*polymorphic data*).

1. User calls generic operator to work on polymorphic data.
2. Generic operator dispatches on type. Data is stripped of tag going down.
3. Returned data is tagged with type going up.

Programs that use complex numbers

`add_complex, sub_complex, mul_complex, div_complex`

Complex Numbers Package



Moving across barriers

Issues to think about

- What are the advantages of providing generic operators?
 - Benefits to the user?
 - Impact of future code changes?
- What are the pros/cons of dispatching on type.
 - What changes are needed when a new representation is installed?
 - Does it resolve the name conflict?

Problems with Dispatch on Type

- Generic operators need to know all the types available.
- Adding a new type means changing all operators to dispatch correctly.
- Does not resolve name conflict.

Strategy #2: Data-directed programming

- How about using a table and doing a table lookup?
- Generic operator look at tag on data and find the correct operation from table
- Address problem of naming conflicts
- Allows easy extension: just add more entries to the table!

Table of operations

		Type	
		Polar	Rectangular
Operations	real_part	real_part	real_part
	imag_part	imag_part	imag_part
	magnitude	magnitude	magnitude
	angle	angle	angle

In creating generic operators, we were really selecting the appropriate lower-level function based on operation and type, as summarized in table above.

Table manipulation

- Suppose we have the following:
 - `put(<op>, <type>, <item>)`
installs <item> in table, indexed by <op> and <type>
 - `get(<op>, <type>)`
looks up <op>, <type> entry in table and returns item found there.

Python Dictionary

- Often convenient to have a data structure that allow retrieval by keyword, i.e. put + get
- Table of **key-value pairs**. Commonly called Associative arrays
- Python dictionaries use the curly braces { }

Python Dictionary

```
{key1:value1, key2:value2, ...}
```

```
>>> {} # empty dict
```

```
>>> weather = {'temp':{2:26.8,  
14:31.1}, 'wind':0, 'desc':'cloudy'}
```

is equivalent to

```
>>> weather = dict(temp={2:26.8,  
14:31.1}, wind=0, desc='cloudy')
```

Python Dictionary

more dictionary constructors:

```
>>> dict([(1, 2), (2, 4), (3, 6)])  
{1:2, 2:4, 3:6}
```

```
>>> weather = {'temp':{2:26.8,  
14:31.1}, 'wind':0, 'desc':'cloudy'}
```

```
>>> weather['temp'] ← Accessed using  
{2:26.8, 14:31.1} key
```

```
>>> weather['tomorrow']
```

```
KeyError: 'tomorrow'
```

Python Dictionary

```
>>> 'wind' in weather
True
>>> 0 in weather
False
>>> weather['is_nice'] = True # adds an
                                # entry
>>> del weather['temp'] # delete an entry
>>> list(weather.keys())
['desc', 'is_nice', 'wind']
>>> list(weather.values())
['cloudy', True, 0]
```

Checks if
key exists

Looping construct

```
>>> for key in weather:  
    print(weather[key])
```

```
cloudy
```

```
True
```

```
0
```

```
>>> for key, value in weather.items():  
    print(key, value)
```

```
desc cloudy
```

```
is_nice True
```

```
wind 0
```

```
>>> weather.clear() # delete all entries
```


Table implementation

```
def put(op, type, proc):  
    if op not in procs:  
        procs[op] = {}  
    procs[op][type] = proc
```

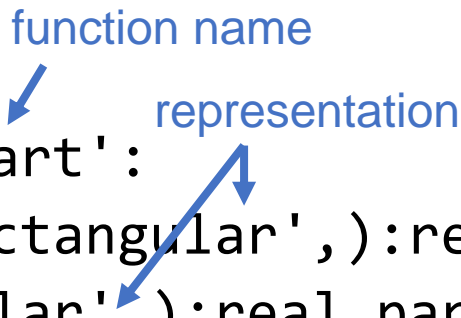
```
def get(op, type):  
    return procs[op][type]
```

	Types	
	Polar	Rectangular
Operations	real_part	real_part
	imag_part	imag_part
	magnitude	magnitude
	angle	angle

2D associative array

Resulting Table

```
procs =  
{ 'real_part':  
    { ('rectangular',): real_part_rectangular,  
      ('polar',): real_part_polar },  
  'imag_part':  
    { ('rectangular',): imag_part_rectangular,  
      ('polar',): imag_part_polar },  
  'magnitude':  
    { ('rectangular',): magnitude_rectangular,  
      ('polar',): magnitude_polar },  
  'angle':  
    { ('rectangular',): angle_rectangular,  
      ('polar',): angle_polar } }
```



Installing Wai Kay's package

```
def install_rectangular-package():  
    # internal functions  
    # unchanged from Wai Kay's initial code  
    def make_from_real_imag(x, y):  
        return attach_tag('rectangular',(x, y))  
    def real_part(z):  
        return z[1]  
    def imag_part(z):  
        return z[2]  
    def magnitude(z):  
        return math.hypot(real_part(z),imag_part(z))  
    def angle(z):  
        return math.atan(imag_part(z)/ real_part(z))  
    def make_from_mag_ang(r, a)  
        return make_from_real_imag (r *  
math.cos(a), r *math.sin(a))
```

Installing Wai Kay's package

```
# cont'd
# interface to the rest of the system
def tag(x):
    return attach_tag('rectangular', x)

put('real_part', ('rectangular', ), real_part)
put('imag_part', ('rectangular', ), imag_part)
put('magnitude', ('rectangular', ), magnitude)
put('angle', ('rectangular', ), angle)
put('make_from_real_imag', 'rectangular',
    make_from_real_imag)
put('make_from_mag_ang', 'rectangular',
    make_from_mag_ang)

return 'done'
```

Installing Karen's package

```
def install_polar-package():  
    # internal functions  
    # unchanged from Karen's original code  
    def make_from_mag_ang(r, a):  
        return attach_tag('polar', (r, a))  
    def magnitude(z):  
        return z[1]  
    def angle(z):  
        return z[2]  
    def real_part(z):  
        return magnitude(z) * math.cos(angle(z))  
    def imag_part(z):  
        return magnitude(z) * math.sin(angle(z))  
    def make_from_real_imag(x, y):  
        return make_from_mag_ang(math.hypot(x, y),  
                                   math.atan(y / x))
```

Installing Karen's package

```
# cont'd
# interface to the rest of the system
def tag(x):
    return attach_tag('polar', x)

put('real_part', ('polar', ), real_part)
put('imag_part', ('polar', ), imag_part)
put('magnitude', ('polar', ), magnitude)
put('angle', ('polar', ), angle)
put('make_from_real_imag', 'polar', make_from_real_imag)
put('make_from_mag_ang', 'polar', make-from-mag-ang)

return 'done'
```

Observation

- No name conflicts!
 - Because all function names are internal (local) to the installer.
 - Thus, each programmer can use identical names.
 - Names live in separate *name space*.
- Installer defines all operators and places them in the table according to operation and type.
- Tags still required to distinguish between types.

Making generic operators

```
def apply_generic(op, *args):  
    type_tags = tuple(map(type_tag, args))  
    proc = get(op, type_tags)  
    return proc(*args)  
  
def type_tag(datum):  
    if type(datum) == tuple  
        and len(datum) == 3:  
        return datum[0]  
    else:  
        raise Exception('Bad tagged datum --  
                        type_tag' + datum)
```


The * notation

- For variable number of arguments

```
def f(x, y, *z):  
    <body>
```

- function `f` can be called with 2, or more arguments.
- Calling `f(1, 2, 3, 4, 5, 6)`: in the body,
 - `x` \rightarrow 1,
 - `y` \rightarrow 2,
 - `z` \rightarrow (3, 4, 5, 6)

The * notation

- You can also call a function for which you don't know in advance how many arguments there will be using *.

```
def funky(op, args):  
    return op(*args)
```

```
print(funky(lambda x: x*x, (2,))) → 4
```

```
print(funky(lambda x,y: x+y, (2, 1))) → 3
```

Making generic operators

```
def apply_generic(op, *args):  
    type_tags = tuple(map(type_tag, args))  
    proc = get(op, type_tags)  
    return proc(*args)
```

Example:

```
apply_generic('real_part', z1)  
  op → 'real_part'  
  args → (z1)  
  type_tags → ('polar',)  
  proc → get('real_part', ('polar',)) → lambda...  
  lambda...(z1)
```

Making generic operators

```
def apply_generic(op, *args):  
    type_tags = tuple(map(type_tag, args))  
    proc = get(op, type_tags)  
    return proc(*args)  
  
def real_part(z):  
    return apply_generic('real_part', z)  
  
def imag_part(z):  
    return apply_generic('imag_part', z)  
  
def magnitude(z):  
    return apply_generic('magnitude', z)  
  
def angle(z):  
    return apply_generic('angle', z)
```

Note:

- `apply_generic` retrieves appropriate function `proc` from table based on `<op>`, `<type>`.
- Then applies `proc` to list of arguments.
- This style of using a table to dispatch is called data-directed programming.

Issues to think about

What advantages does data-directed programming have over dispatching on type?

- Consider the effect of adding a new type or new operation. Any code changes?
- Any name conflicts?
- Who should maintain table?

Generic Arithmetic

- We can push this idea further!
- We can use generic operators to handle *completely different data*, not just different representations of the same data.
- Let's build a **generic arithmetic** package.
 - Works with rational numbers, complex numbers, ordinary numbers, even polynomials!
 - Provide generic operators: add, sub, mul, div

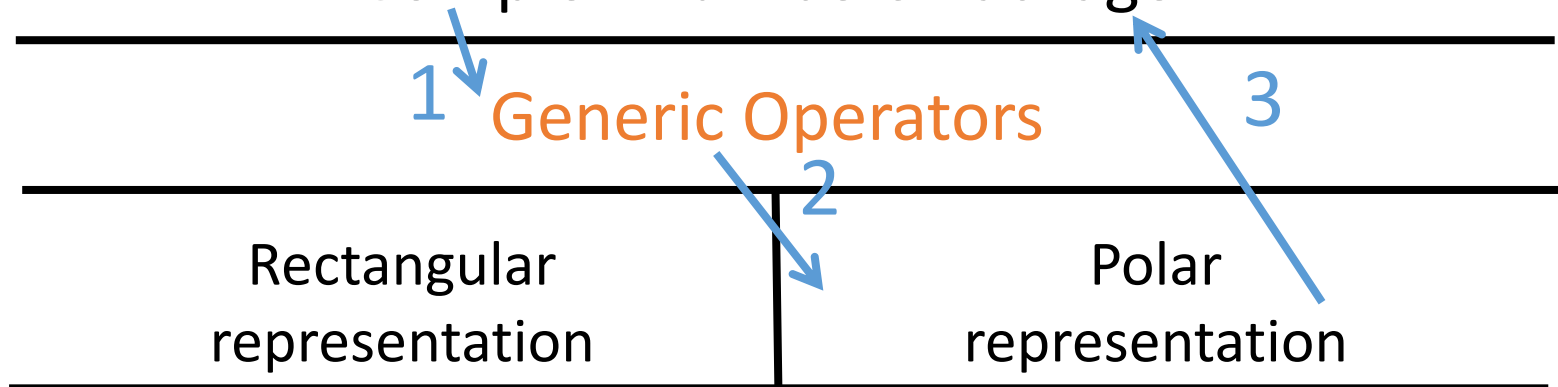
Complex Numbers

1. User calls generic operator to work on polymorphic data.
2. Generic operator dispatches on type. Data is stripped of tag going down.
3. Returned data is tagged with type going up.

Programs that use complex numbers

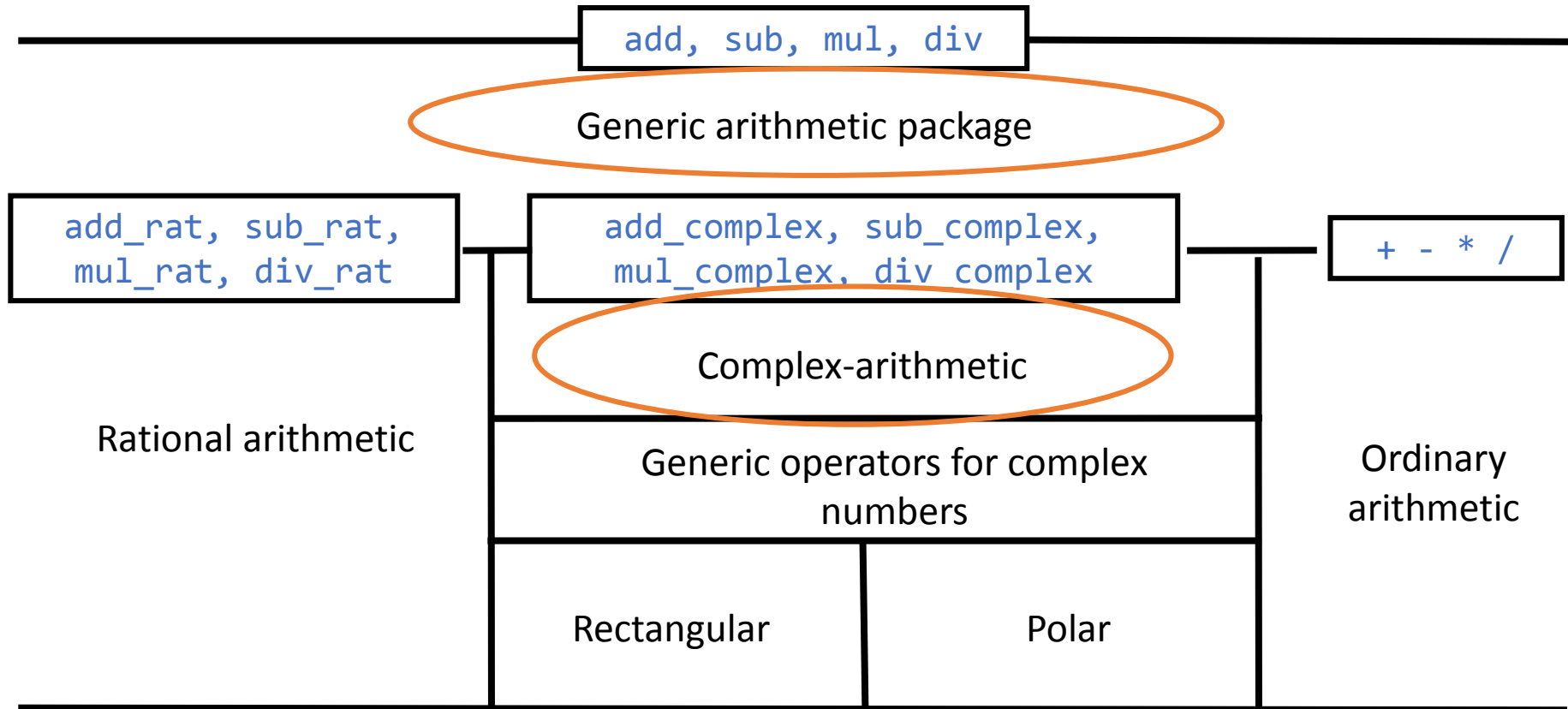
`add_complex, sub_complex, mul_complex, div_complex`

Complex Numbers Package



Generic Arithmetic

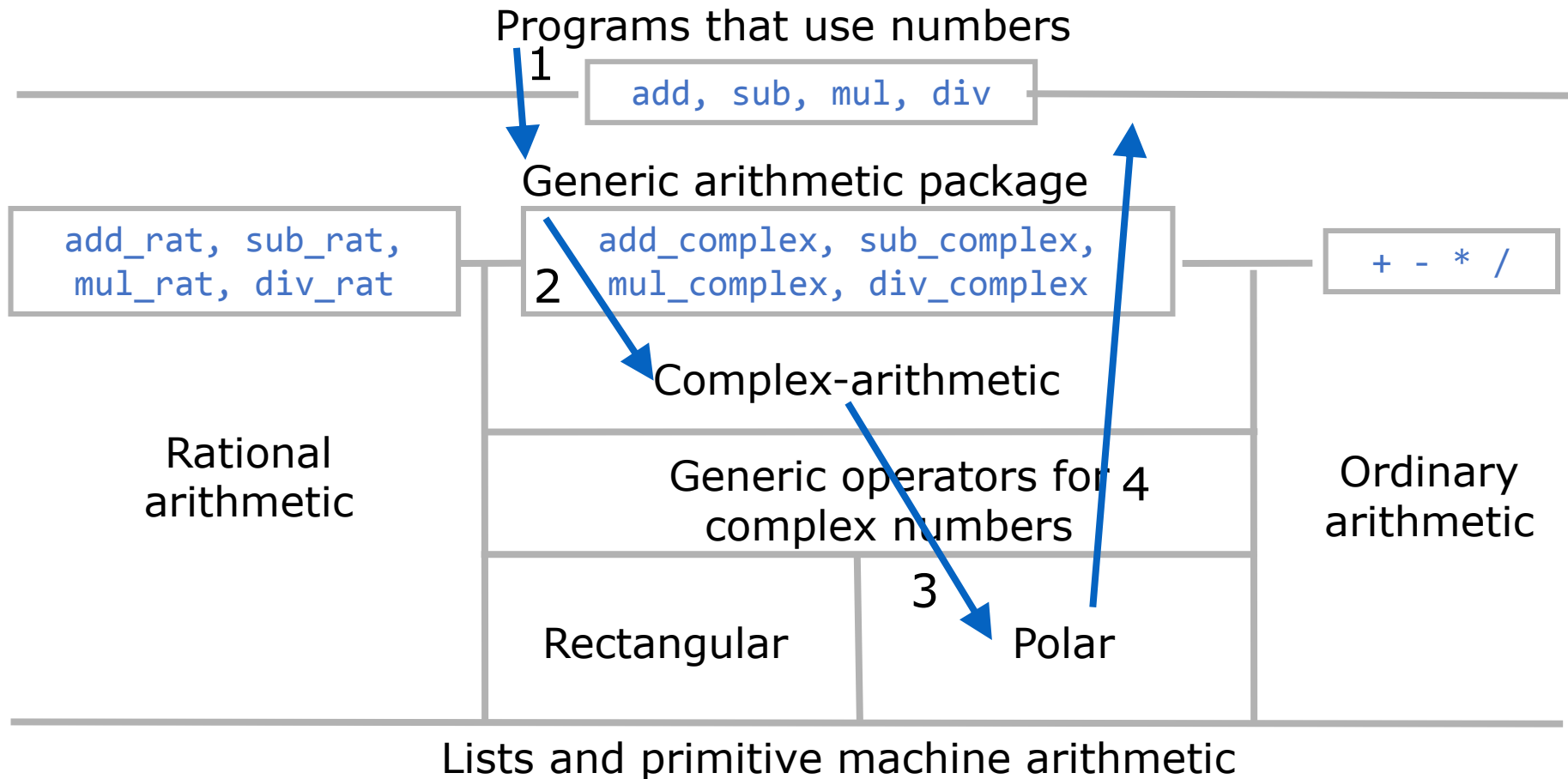
Programs that use numbers



Lists and primitive machine arithmetic

Generic Arithmetic

1. User calls generic arithmetic operators to work on polymorphic numbers.
2. Generic operator dispatches on type. Data is **stripped of tag** going down, e.g. complex
3. Complex package calls generic operators for rect/polar forms. Tag is stripped for data going down.
4. Returned data **tagged with types** going up: polar, and complex.



Implementation strategy

Use data-directed
programming, rather than
dispatch on type

Generic Arithmetic

```
def install_complex_package():
```

```
# imported functions from rectangular and polar packages
```

```
def make_from_real_imag(x, y):
```

```
return get('make_from_real_imag', 'rectangular')(x, y)
```

```
def make_from_mag_ang(r, a):
```

```
return get('make_from_mag_ang', 'polar')(r, a)
```

internal functions

```
def add_complex(z1, z2):
```

```
return make_from_real_imag(real_part(z1) + real_part(z2),
                           imag_part(z1) + imag_part(z2))
```

```
def sub_complex(z1, z2):
```

```
return make_from_real_imag(real_part(z1) - real_part(z2),
                           imag_part(z1) - imag_part(z2))
```

```
def mul_complex(z1, z2):
```

```
return make_from_mag_ang(magnitude(z1) * magnitude(z2),
                          angle(z1) + angle(z2))
```

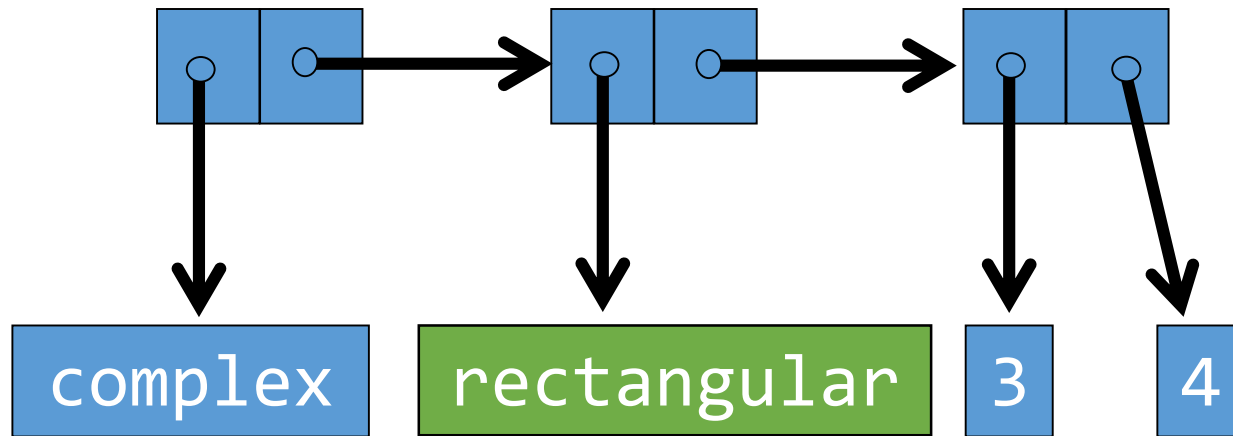
```
def div_complex(z1, z2):
```

```
return make_from_mag_ang(magnitude(z1) / magnitude(z2),
                        angle(z1) - angle(z2))
```

Generic Arithmetic

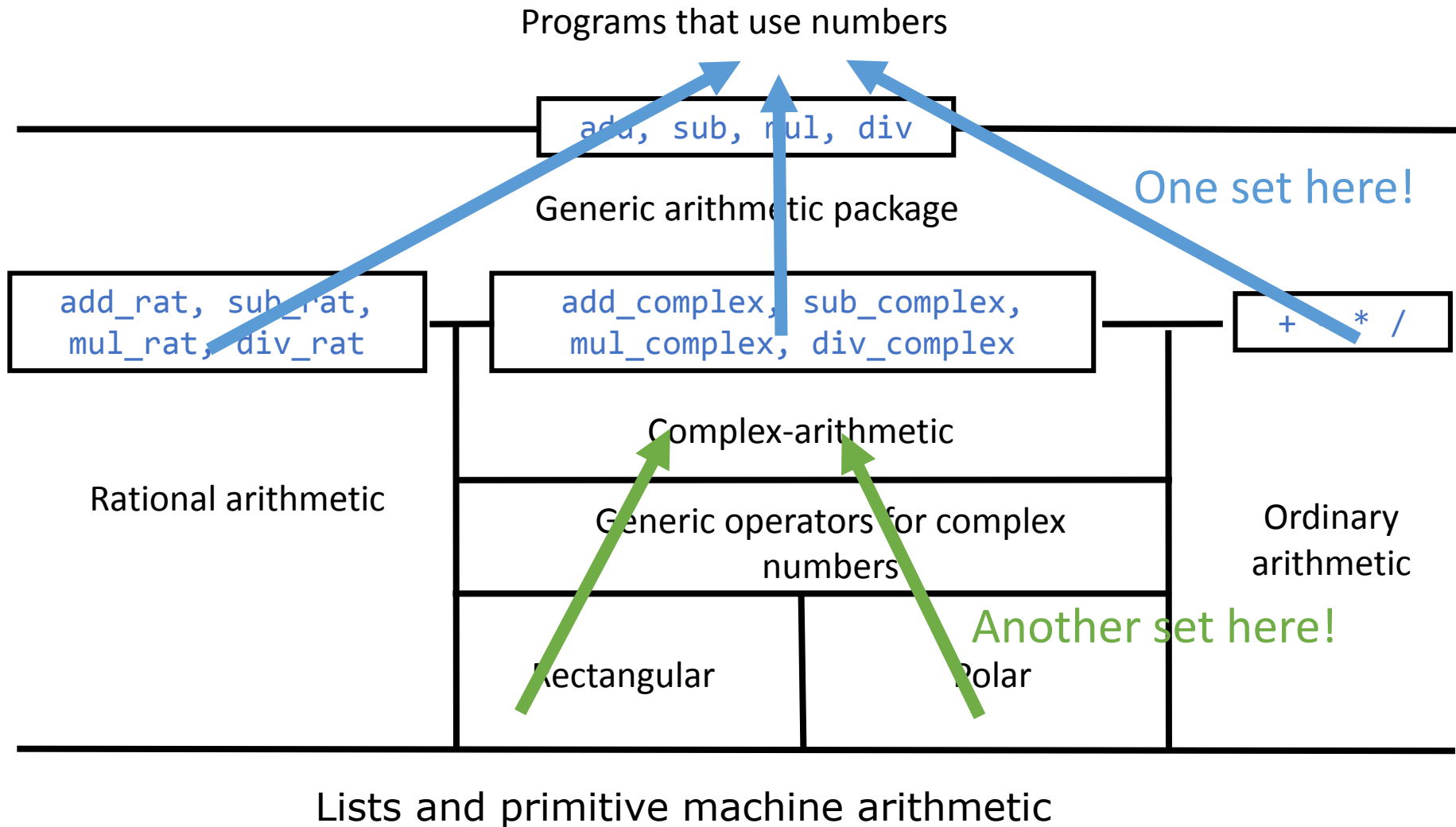
```
# interface to rest of the system
def tag(z):
    return attach_tag('complex', z)
put('add', ('complex', 'complex'),
    lambda z1, z2: tag(add_complex(z1, z2)))
put('sub', ('complex', 'complex'),
    lambda z1, z2: tag(sub_complex(z1, z2)))
put('mul', ('complex', 'complex'),
    lambda z1, z2: tag(mul_complex(z1, z2)))
put('div', ('complex', 'complex'),
    lambda z1, z2: tag(div_complex(z1, z2)))
put('make_from_real_imag', 'complex',
    lambda x, y: tag(make_from_real_imag(x, y)))
put('make_from_mag_ang', 'complex',
    lambda x, y: tag(make_from_mag_ang (x, y)))
return 'done'
```

Tags upon tags



- There are actually two tags
- Why two tags?

Generic Arithmetic



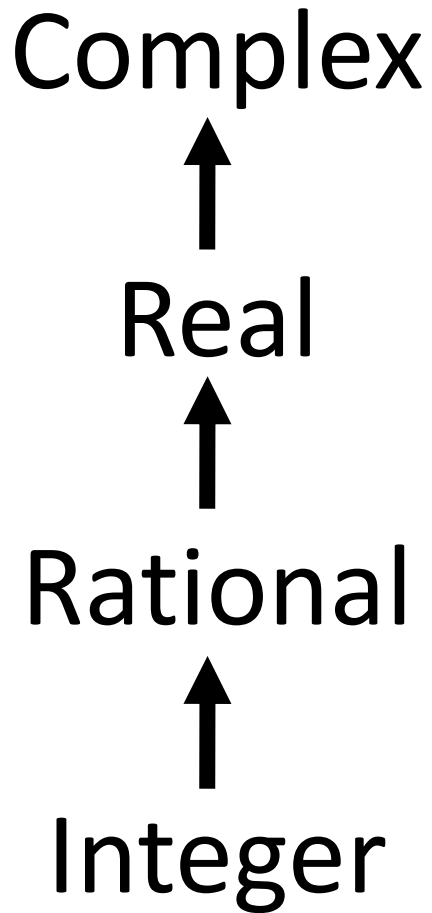
Layering Abstraction

- This is an **IMPORTANT(!)** idea:
 - As data passes down a layer, a tag is added
 - As data passes up a layer, the tag is stripped away
- Applicable to other areas of computer science and not just data abstraction
- **Computer Networking!**

Different data types

- So far, we are adding numbers of the same type: complex + complex.
- What happens if we need to add complex to rational?
- Answer: coercion
- Force (or convert) one data type into another.
- Insight: Exploit the hierarchy of numbers.

Number Hierarchy



- Thus we can coerce a number of a lower type to one of a higher type.
- But the reverse may not always be possible!
 - e.g. $3+4i$ to integer?

All seems
like a blur?

Missions 11 and 12 will fix
that. 😊

Strategy #3: Message passing

- Previous strategies viewed functions as “intelligent”:
 - They dispatch according to type of data.
- In *message passing*, it is the data that is “intelligent”:
 - They know how to act on themselves.
 - You just “tell” the data what you want.

Message passing

```
def make_from_real_imag(x, y):  
    def dispatch(op):  
        if op == 'real_part':  
            return x  
        elif op == 'imag_part':  
            return y  
        elif op == 'magnitude':  
            return math.hypot(x, y)  
        elif op == 'angle':  
            return math.atan(y / x)  
        else:  
            raise Exception("Unknown op --  
                             make_from_real_imag" + op)  
    return dispatch
```

Message passing

- “Data” is the function `dispatch`.
- `dispatch` accepts a “message” op, and performs the necessary action according to the op.

```
def real_part(z):  
    return z('real_part')  
def imag_part(z):  
    return z('imag_part')
```

Message passing

- This idea of “passing a message” to the data and letting the data do the job is the basis of **object-oriented programming**.
 - e.g. Java, C++, Smalltalk
 - Data are objects.
 - Functions are actions that objects perform.
 - Objects “talk” to other objects by passing messages

Summary

- Challenges of managing Multiple Representations
 - Matching data types to operations.
 - Resolving name conflicts.
 - Managing changes/future modifications
- Abstraction layer between the underlying representations and the user

Summary

- You have now seen 3 strategies for creating generic operators:
 - Dispatching on type
 - Data-directed programming
 - Message passing
- What are the pros/cons of each?
 - How does each resolve name conflicts?
 - What happens when a new type/operation is added?